

Built In Functions

Table of contents

1 Introduction.....	2
2 Dynamic Invokers.....	2
3 Eval Functions.....	3
4 Load/Store Functions.....	14
5 Math Functions.....	22
6 String Functions.....	32
7 Tuple, Bag, Map Functions.....	39

1. Introduction

Pig comes with a set of built in functions (the eval, load/store, math, string, bag and tuple functions). Two main properties differentiate built in functions from [user defined functions](#) (UDFs). First, built in functions don't need to be registered because Pig knows where they are. Second, built in functions don't need to be qualified when they are used because Pig knows where to find them.

2. Dynamic Invokers

Often you may need to use a simple function that is already provided by standard Java libraries, but for which a [user defined functions](#) (UDF) has not been written. Dynamic invokers allow you to refer to Java functions without having to wrap them in custom UDFs, at the cost of doing some Java reflection on every function call.

```
...
DEFINE UrlDecode InvokeForString('java.net.URLDecoder.decode', 'String
String');
encoded_strings = LOAD 'encoded_strings.txt' as (encoded:chararray);
decoded_strings = FOREACH encoded_strings GENERATE UrlDecode(encoded,
'UTF-8');
...
```

Currently, dynamic invokers can be used for any static function that:

- Accepts no arguments or accepts some combination of strings, ints, longs, doubles, floats, or arrays with these same types
- Returns a string, an int, a long, a double, or a float

Only primitives can be used for numbers; no capital-letter numeric classes can be used as arguments. Depending on the return type, a specific kind of invoker must be used: InvokeForString, InvokeForInt, InvokeForLong, InvokeForDouble, or InvokeForFloat.

The [DEFINE](#) statement is used to bind a keyword to a Java method, as above. The first argument to the InvokeFor* constructor is the full path to the desired method. The second argument is a space-delimited ordered list of the classes of the method arguments. This can be omitted or an empty string if the method takes no arguments. Valid class names are string, long, float, double, and int. Invokers can also work with array arguments, represented in Pig as DataBags of single-tuple elements. Simply refer to string[], for example. Class names are not case sensitive.

The ability to use invokers on methods that take array arguments makes methods like those in org.apache.commons.math.stat.StatUtils available (for processing the results of grouping your datasets, for example). This is helpful, but a word of caution: the resulting UDF will not

be optimized for Hadoop, and the very significant benefits one gains from implementing the Algebraic and Accumulator interfaces are lost here. Be careful if you use invokers this way.

3. Eval Functions

3.1. AVG

Computes the average of the numeric values in a single-column bag.

3.1.1. Syntax

AVG(expression)

3.1.2. Terms

expression	Any expression whose result is a bag. The elements of the bag should be data type int, long, float, or double.
------------	--

3.1.3. Usage

Use the AVG function to compute the average of the numeric values in a single-column bag. AVG requires a preceding GROUP ALL statement for global averages and a GROUP BY statement for group averages.

The AVG function now ignores NULL values.

3.1.4. Example

In this example the average GPA for each student is computed (see the [GROUP](#) operator for information about the field names in relation B).

```
A = LOAD 'student.txt' AS (name:chararray, term:chararray, gpa:float);
DUMP A;
(John,fl,3.9F)
(John,wt,3.7F)
(John,sp,4.0F)
(John,sm,3.8F)
(Mary,fl,3.8F)
(Mary,wt,3.9F)
(Mary,sp,4.0F)
(Mary,sm,4.0F)
```

```

B = GROUP A BY name;

DUMP B;
(John, {(John, fl, 3.9F), (John, wt, 3.7F), (John, sp, 4.0F), (John, sm, 3.8F)})
(Mary, {(Mary, fl, 3.8F), (Mary, wt, 3.9F), (Mary, sp, 4.0F), (Mary, sm, 4.0F)})

C = FOREACH B GENERATE A.name, AVG(A.gpa);

DUMP C;
({(John), (John), (John), (John)}, 3.850000023841858)
({(Mary), (Mary), (Mary), (Mary)}, 3.925000011920929)

```

3.1.5. Types Tables

	int	long	float	double	chararray	bytearray
AVG	long	long	double	double	error	cast as double

3.2. CONCAT

Concatenates two expressions of identical type.

3.2.1. Syntax

```
CONCAT (expression, expression)
```

3.2.2. Terms

expression	Any expression.
------------	-----------------

3.2.3. Usage

Use the CONCAT function to concatenate two expressions. The result values of the two expressions must have identical types.

3.2.4. Example

In this example fields f2 and f3 are concatenated.

```

A = LOAD 'data' as (f1:chararray, f2:chararray, f3:chararray);
DUMP A;

```

```
(apache,open,source)
(hadoop,map,reduce)
(pig,pig,latin)

X = FOREACH A GENERATE CONCAT(f2,f3);

DUMP X;
(opensource)
(mapreduce)
(piglatin)
```

3.3. COUNT

Computes the number of elements in a bag.

3.3.1. Syntax

COUNT(expression)

3.3.2. Terms

expression	An expression with data type bag.
------------	-----------------------------------

3.3.3. Usage

Use the COUNT function to compute the number of elements in a bag. COUNT requires a preceding GROUP ALL statement for global counts and a GROUP BY statement for group counts.

The COUNT function follows syntax semantics and ignores nulls. What this means is that a tuple in the bag will not be counted if the FIRST FIELD in this tuple is NULL. If you want to include NULL values in the count computation, use [COUNT_STAR](#).

Note: You cannot use the tuple designator (*) with COUNT; that is, COUNT(*) will not work.

3.3.4. Example

In this example the tuples in the bag are counted (see the [GROUP](#) operator for information about the field names in relation B).

```
A = LOAD 'data' AS (f1:int,f2:int,f3:int);

DUMP A;
(1,2,3)
```

```

(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)

B = GROUP A BY f1;

DUMP B;
(1, {(1,2,3)})
(4, {(4,2,1), (4,3,3)})
(7, {(7,2,5)})
(8, {(8,3,4), (8,4,3)})

X = FOREACH B GENERATE COUNT(A);

DUMP X;
(1L)
(2L)
(1L)
(2L)

```

3.3.5. Types Tables

	int	long	float	double	chararray	bytearray
COUNT	long	long	long	long	long	long

3.4. COUNT_STAR

Computes the number of elements in a bag.

3.4.1. Syntax

```
COUNT_STAR(expression)
```

3.4.2. Terms

expression	An expression with data type bag.
------------	-----------------------------------

3.4.3. Usage

Use the COUNT_STAR function to compute the number of elements in a bag. COUNT_STAR requires a preceding GROUP ALL statement for global counts and a GROUP BY statement for group counts.

COUNT_STAR includes NULL values in the count computation (unlike [COUNT](#), which ignores NULL values).

3.4.4. Example

In this example COUNT_STAR is used the count the tuples in a bag.

```
X = FOREACH B GENERATE COUNT_STAR(A);
```

3.5. DIFF

Compares two fields in a tuple.

3.5.1. Syntax

DIFF (expression, expression)

3.5.2. Terms

expression	An expression with any data type.
------------	-----------------------------------

3.5.3. Usage

The DIFF function takes two bags as arguments and compares them. Any tuples that are in one bag but not the other are returned in a bag. If the bags match, an empty bag is returned. If the fields are not bags then they will be wrapped in tuples and returned in a bag if they do not match, or an empty bag will be returned if the two records match. The implementation assumes that both bags being passed to the DIFF function will fit entirely into memory simultaneously. If this is not the case the UDF will still function but it will be VERY slow.

3.5.4. Example

In this example DIFF compares the tuples in two bags.

```
A = LOAD 'bag_data' AS
(B1:bag{T1:tuple(t1:int,t2:int)},B2:bag{T2:tuple(f1:int,f2:int)});

DUMP A;
({(8,9),(0,1)},{(8,9),(1,1)})
({(2,3),(4,5)},{(2,3),(4,5)})
({(6,7),(3,7)},{(2,2),(3,7)})

DESCRIBE A;
```

```
a: {B1: {T1: (t1: int,t2: int)},B2: {T2: (f1: int,f2: int)}}
X = FOREACH A DIFF(B1,B2);

grunt> dump x;
({(0,1),(1,1)})
({})
({(6,7),(2,2)})
```

3.6. IsEmpty

Checks if a bag or map is empty.

3.6.1. Syntax

IsEmpty(expression)

3.6.2. Terms

expression	An expression with any data type.
------------	-----------------------------------

3.6.3. Usage

The IsEmpty function checks if a bag or map is empty (has no data). The function can be used to filter data.

3.6.4. Example

In this example all students with an SSN but no name are located.

```
SSN = load 'ssn.txt' using PigStorage() as (ssn:long);

SSN_NAME = load 'students.txt' using PigStorage() as (ssn:long,
name:chararray);

/* do a left outer join of SSN with SSN_Name */
X = JOIN SSN by ssn LEFT OUTER, SSN_NAME by ssn;

/* only keep those ssn's for which there is no name */
Y = filter X by IsEmpty(SSN_NAME);
```

3.7. MAX

Computes the maximum of the numeric values or chararrays in a single-column bag. MAX requires a preceding GROUP ALL statement for global maximums and a GROUP BY

statement for group maximums.

3.7.1. Syntax

MAX(expression)

3.7.2. Terms

expression	An expression with data types int, long, float, double, or chararray.
------------	---

3.7.3. Usage

Use the MAX function to compute the maximum of the numeric values or chararrays in a single-column bag.

3.7.4. Example

In this example the maximum GPA for all terms is computed for each student (see the GROUP operator for information about the field names in relation B).

```
A = LOAD 'student' AS (name:chararray, session:chararray, gpa:float);
DUMP A;
(John,fl,3.9F)
(John,wt,3.7F)
(John,sp,4.0F)
(John,sm,3.8F)
(Mary,fl,3.8F)
(Mary,wt,3.9F)
(Mary,sp,4.0F)
(Mary,sm,4.0F)

B = GROUP A BY name;
DUMP B;
(John, {(John,fl,3.9F), (John,wt,3.7F), (John,sp,4.0F), (John,sm,3.8F)})
(Mary, {(Mary,fl,3.8F), (Mary,wt,3.9F), (Mary,sp,4.0F), (Mary,sm,4.0F)})

X = FOREACH B GENERATE group, MAX(A.gpa);
DUMP X;
(John,4.0F)
(Mary,4.0F)
```

3.7.5. Types Tables

	int	long	float	double	chararray	bytearray
MAX	int	long	float	double	chararray	cast as double

3.8. MIN

Computes the minimum of the numeric values or chararrays in a single-column bag. MIN requires a preceding GROUP... ALL statement for global minimums and a GROUP ... BY statement for group minimums.

3.8.1. Syntax

```
MIN(expression)
```

3.8.2. Terms

expression	An expression with data types int, long, float, double, or chararray.
------------	---

3.8.3. Usage

Use the MIN function to compute the minimum of a set of numeric values or chararrays in a single-column bag.

3.8.4. Example

In this example the minimum GPA for all terms is computed for each student (see the GROUP operator for information about the field names in relation B).

```
A = LOAD 'student' AS (name:chararray, session:chararray, gpa:float);
DUMP A;
(John,fl,3.9F)
(John,wt,3.7F)
(John,sp,4.0F)
(John,sm,3.8F)
(Mary,fl,3.8F)
(Mary,wt,3.9F)
(Mary,sp,4.0F)
(Mary,sm,4.0F)
B = GROUP A BY name;
```

```
DUMP B;
(John, {(John, fl, 3.9F), (John, wt, 3.7F), (John, sp, 4.0F), (John, sm, 3.8F)})
(Mary, {(Mary, fl, 3.8F), (Mary, wt, 3.9F), (Mary, sp, 4.0F), (Mary, sm, 4.0F)})

X = FOREACH B GENERATE group, MIN(A.gpa);

DUMP X;
(John, 3.7F)
(Mary, 3.8F)
```

3.8.5. Types Tables

	int	long	float	double	chararray	bytearray
MIN	int	long	float	double	chararray	cast as double

3.9. SIZE

Computes the number of elements based on any Pig data type.

3.9.1. Syntax

```
SIZE(expression)
```

3.9.2. Terms

expression	An expression with any data type.
------------	-----------------------------------

3.9.3. Usage

Use the SIZE function to compute the number of elements based on the data type (see the Types Tables below). SIZE includes NULL values in the size computation. SIZE is not algebraic.

3.9.4. Example

In this example the number of characters in the first field is computed.

```
A = LOAD 'data' as (f1:chararray, f2:chararray, f3:chararray);
(apache, open, source)
(hadoop, map, reduce)
(pig, pig, latin)
```

```
X = FOREACH A GENERATE SIZE(f1);
DUMP X;
(6L)
(6L)
(3L)
```

3.9.5. Types Tables

int	returns 1
long	returns 1
float	returns 1
double	returns 1
chararray	returns number of characters in the array
bytearray	returns number of bytes in the array
tuple	returns number of fields in the tuple
bag	returns number of tuples in bag
map	returns number of key/value pairs in map

3.10. SUM

Computes the sum of the numeric values in a single-column bag. SUM requires a preceding GROUP ALL statement for global sums and a GROUP BY statement for group sums.

3.10.1. Syntax

```
SUM(expression)
```

3.10.2. Terms

expression	An expression with data types int, long, float, double, or bytearray cast as double.
------------	--

--	--

3.10.3. Usage

Use the SUM function to compute the sum of a set of numeric values in a single-column bag.

3.10.4. Example

In this example the number of pets is computed. (see the GROUP operator for information about the field names in relation B).

```
A = LOAD 'data' AS (owner:chararray, pet_type:chararray, pet_num:int);
DUMP A;
(Alice,turtle,1)
(Alice,goldfish,5)
(Alice,cat,2)
(Bob,dog,2)
(Bob,cat,2)

B = GROUP A BY owner;
DUMP B;
(Alice,{(Alice,turtle,1),(Alice,goldfish,5),(Alice,cat,2)})
(Bob,{(Bob,dog,2),(Bob,cat,2)})

X = FOREACH B GENERATE group, SUM(A.pet_num);
DUMP X;
(Alice,8L)
(Bob,4L)
```

3.10.5. Types Tables

	int	long	float	double	chararray	bytearray
SUM	long	long	double	double	error	cast as double

3.11. TOKENIZE

Splits a string and outputs a bag of words.

3.11.1. Syntax

TOKENIZE(expression [, 'field_delimiter'])
--

3.11.2. Terms

expression	An expression with data type chararray.
'field_delimiter'	An optional field delimiter (in single quotes). If field_delimiter is null or not passed, the following will be used as delimiters: space [], double quote ["], coma [,] parenthesis [()], star [*].

3.11.3. Usage

Use the TOKENIZE function to split a string of words (all words in a single tuple) into a bag of words (each word in a single tuple).

3.11.4. Example

In this example the strings in each row are split.

```
A = LOAD 'data' AS (f1:chararray);

DUMP A;
(Here is the first string.)
(Here is the second string.)
(Here is the third string.)

X = FOREACH A GENERATE TOKENIZE(f1);

DUMP X;
({(Here),(is),(the),(first),(string.)})
({(Here),(is),(the),(second),(string.)})
({(Here),(is),(the),(third),(string.)})
```

In this example a field delimiter is specified.

```
{code}
A = LOAD 'data' AS (f1:chararray);
B = FOREACH A TOKENIZE (f1,'||');
DUMP B;
{code}
```

4. Load/Store Functions

Load/store functions determine how data goes into Pig and comes out of Pig. Pig provides a set of built-in load/store functions, described in the sections below. You can also write your own load/store functions (see [User Defined Functions](#)).

4.1. Handling Compression

Support for compression is determined by the load/store function. PigStorage and TextLoader support gzip and bzip compression for both read (load) and write (store). BinStorage does not support compression.

To work with gzip compressed files, input/output files need to have a .gz extension. Gzipped files cannot be split across multiple maps; this means that the number of maps created is equal to the number of part files in the input location.

```
A = load 'myinput.gz';  
store A into 'myoutput.gz';
```

To work with bzip compressed files, the input/output files need to have a .bz or .bz2 extension. Because the compression is block-oriented, bziped files can be split across multiple maps.

```
A = load 'myinput.bz';  
store A into 'myoutput.bz';
```

Note: PigStorage and TextLoader correctly read compressed files as long as they are NOT CONCATENATED FILES generated in this manner:

- `cat *.gz > text/concat.gz`
- `cat *.bz > text/concat.bz`
- `cat *.bz2 > text/concat.bz2`

If you use concatenated gzip or bzip files with your Pig jobs, you will NOT see a failure but the results will be INCORRECT.

4.2. BinStorage

Loads and stores data in machine-readable format.

4.2.1. Syntax

BinStorage()

4.2.2. Terms

none	no parameters
------	---------------

4.2.3. Usage

Pig uses BinStorage to load and store the temporary data that is generated between multiple MapReduce jobs.

- BinStorage works with data that is represented on disk in machine-readable format. BinStorage does NOT support [compression](#).
- BinStorage supports multiple locations (files, directories, globs) as input.

Occasionally, users use BinStorage to store their own data. However, because BinStorage is a proprietary binary format, the original data is never in BinStorage - it is always a derivation of some other data.

We have seen several examples of users doing something like this:

```
a = load 'b.txt' as (id, f);
b = group a by id;
store b into 'g' using BinStorage();
```

And then later:

```
a = load 'g/part*' using BinStorage() as (id, d:bag{t:(v, s)});
b = foreach a generate (double)id, flatten(d);
dump b;
```

There is a problem with this sequence of events. The first script does not define data types and, as the result, the data is stored as a bytearray and a bag with a tuple that contains two bytearrays. The second script attempts to cast the bytearray to double; however, since the data originated from a different loader, it has no way to know the format of the bytearray or how to cast it to a different type. To solve this problem, Pig:

- Sends an error message when the second script is executed: "ERROR 1118: Cannot cast bytes loaded from BinStorage. Please provide a custom converter."
- Allows you to use a custom converter to perform the casting.

```
a = load 'g/part*' using BinStorage('Utf8StorageConverter') as (id,
d:bag{t:(v, s)});
b = foreach a generate (double)id, flatten(d);
dump b;
```

4.2.4. Examples

In this example BinStorage is used with the LOAD and STORE functions.

```
A = LOAD 'data' USING BinStorage();
```

```
STORE X into 'output' USING BinStorage();
```

In this example BinStorage is used to load multiple locations.

```
A = LOAD 'input1.bin, input2.bin' USING BinStorage();
```

BinStorage does not track data lineage. When Pig uses BinStorage to move data between MapReduce jobs, Pig can figure out the correct cast function to use and apply it. However, as shown in the example below, when you store data using BinStorage and then use a separate Pig Latin script to read data (thus losing the type information), it is your responsibility to correctly cast the data before storing it using BinStorage.

```
raw = load 'sampledata' using BinStorage() as (col1,col2, col3);
--filter out null columns
A = filter raw by col1#'bcookie' is not null;

B = foreach A generate col1#'bcookie' as reqcolumn;
describe B;
--B: {reqcolumn: bytearray}
X = limit B 5;
dump X;
(36co9b55onr8s)
(36co9b55onr8s)
(36hilul5oolq1)
(36hilul5oolq1)
(36l4cj15ooa8a)

B = foreach A generate (chararray)col1#'bcookie' as convertedcol;
describe B;
--B: {convertedcol: chararray}
X = limit B 5;
dump X;
()
()
()
()
()
```

4.3. JsonLoader, JsonStorage

Load or store JSON data.

4.3.1. Syntax

```
JsonLoader(['schema'])
```

```
JsonStorage()
```

4.3.2. Terms

schema	An optional Pig schema, in single quotes.
--------	---

4.3.3. Usage

Use `JsonLoader` to load JSON data.

Use `JsonStorage` to store JSON data.

Note that there is no concept of delimit in `JsonLoader` or `JsonStorer`. The data is encoded in standard JSON format. `JsonLoader` optionally takes a schema as the construct argument.

4.3.4. Examples

In this example data is loaded with a schema.

```
a = load 'a.json' using
  JsonLoader('a0:int,a1:{(a10:int,a11:chararray)},a2:(a20:double,a21:bytearray),a3:[chara
```

In this example data is loaded without a schema; it assumes there is a `.pig_schema` (produced by `JsonStorage`) in the input directory.

```
a = load 'a.json' using JsonLoader();
```

4.4. PigDump

Stores data in UTF-8 format.

4.4.1. Syntax

<code>PigDump()</code>

4.4.2. Terms

none	no parameters
------	---------------

4.4.3. Usage

`PigDump` stores data as tuples in human-readable UTF-8 format.

4.4.4. Example

In this example PigDump is used with the STORE function.

```
STORE X INTO 'output' USING PigDump();
```

4.5. PigStorage

Loads and stores data as structured text files.

4.5.1. Syntax

```
PigStorage( [field_delimiter] , ['options'] )
```

4.5.2. Terms

field_delimiter	The default field delimiter is tab ('\t'). You can specify other characters as field delimiters; however, be sure to encase the characters in single quotes.
'options'	A string that contains space-separated options ('optionA optionB optionC') Currently supported options are: <ul style="list-style-type: none">• ('schema') - Stores the schema of the relation using a hidden JSON file.• ('noschema') - Ignores a stored schema during the load.• ('tagsource') - Add a first column indicates the input file of the record.

4.5.3. Usage

PigStorage is the default function used by Pig to load/store the data. PigStorage supports structured text files (in human-readable UTF-8 format) in compressed or uncompressed form (see [Handling Compression](#)). All Pig [data types](#) (both simple and complex) can be read/written using this function. The input data to the load can be a file, a directory or a glob.

Load/Store Statements

Load statements – PigStorage expects data to be formatted using field delimiters, either the tab character ('\t') or other specified character.

Store statements – PigStorage outputs data using field delimiters, either the tab character ('\t') or other specified character, and the line feed record delimiter ('\n').

Field/Record Delimiters

Field Delimiters – For load and store statements the default field delimiter is the tab character ('\t'). You can use other characters as field delimiters, but separators such as ^A or Ctrl-A should be represented in Unicode (\u0001) using UTF-16 encoding (see Wikipedia [ASCII](#), [Unicode](#), and [UTF-16](#)).

Record Delimiters – For load statements Pig interprets the line feed ('\n'), carriage return ('\r' or CTRL-M) and combined CR + LF ('\r\n') characters as record delimiters (do not use these characters as field delimiters). For store statements Pig uses the line feed ('\n') character as the record delimiter.

Schemas

If the schema option is specified, a hidden ".pig_schema" file is created in the output directory when storing data. It is used by PigStorage (with or without -schema) during loading to determine the field names and types of the data without the need for a user to explicitly provide the schema in an as clause, unless noschema is specified. No attempt to merge conflicting schemas is made during loading. The first schema encountered during a file system scan is used.

Additionally, if the schema option is specified, a ".pig_headers" file is created in the output directory. This file simply lists the delimited aliases. This is intended to make export to tools that can read files with header lines easier (just cat the header to your data).

If the schema option is NOT specified, a schema will not be written when storing data.

If the noschema option is NOT specified, and a schema is found, it gets loaded when loading data.

Note that regardless of whether or not you store the schema, you always need to specify the correct delimiter to read your data. If you store reading delimiter "#" and then load using the default delimiter, your data will not be parsed correctly.

Record Provenance

If tagsource option is specified, PigStorage will add a pseudo-column INPUT_FILE_NAME to the beginning of the record. As the name suggests, it is the input file name containing this particular record.

Complex Data Types

The formats for complex data types are shown here:

- **Tuple**: enclosed by (), items separated by ","
 - Non-empty tuple: (item1,item2,item3)
 - Empty tuple is valid: ()
- **Bag**: enclosed by {}, tuples separated by ","
 - Non-empty bag: {code} {(tuple1),(tuple2),(tuple3)} {code}
 - Empty bag is valid: {}
- **Map**: enclosed by [], items separated by ",", key and value separated by "#"
 - Non-empty map: [key1#value1,key2#value2]
 - Empty map is valid: []

If load statement specify a schema, Pig will convert the complex type according to schema. If conversion fails, the affected item will be null (see [Nulls and Pig Latin](#)).

4.5.4. Examples

In this example PigStorage expects input.txt to contain tab-separated fields and newline-separated records. The statements are equivalent.

```
A = LOAD 'student' USING PigStorage('\t') AS (name: chararray, age:int, gpa: float);
```

```
A = LOAD 'student' AS (name: chararray, age:int, gpa: float);
```

In this example PigStorage stores the contents of X into files with fields that are delimited with an asterisk (*). The STORE function specifies that the files will be located in a directory named output and that the files will be named part-nnnnn (for example, part-00000).

```
STORE X INTO 'output' USING PigStorage('*');
```

In this example, PigStorage loads data with complex data type, a bag of map and double.

```
a = load '1.txt' as (a0:{t:(m:map[int],d:double)});  
{([foo#1,bar#2],34.0),([white#3,yellow#4],45.0)} : valid  
{([foo#badint],baddouble)} : conversion fail for badint/baddouble, get  
{([foo#],,)}  
{ } : valid, empty bag
```

4.6. TextLoader

Loads unstructured data in UTF-8 format.

4.6.1. Syntax

TextLoader()

4.6.2. Terms

none	no parameters
------	---------------

4.6.3. Usage

TextLoader works with unstructured data in UTF8 format. Each resulting tuple contains a single field with one line of input text. TextLoader also supports [compression](#).

Currently, TextLoader support for compression is limited.

TextLoader cannot be used to store data.

4.6.4. Example

In this example TextLoader is used with the LOAD function.

```
A = LOAD 'data' USING TextLoader();
```

5. Math Functions

For general information about these functions, see the [Java API Specification](#), [Class Math](#).

Note the following:

- Pig function names are case sensitive and UPPER CASE.
- Pig may process results differently than as stated in the Java API Specification:
 - If the result value is null or empty, Pig returns null.
 - If the result value is not a number (NaN), Pig returns null.
 - If Pig is unable to process the expression, Pig returns an exception.

5.1. ABS

Returns the absolute value of an expression.

5.1.1. Syntax

ABS(expression)

5.1.2. Terms

expression	Any expression whose result is type int, long, float, or double.
------------	--

5.1.3. Usage

Use the ABS function to return the absolute value of an expression. If the result is not negative ($x \neq 0$), the result is returned. If the result is negative ($x < 0$), the negation of the result is returned.

5.2. ACOS

Returns the arc cosine of an expression.

5.2.1. Syntax

ACOS(expression)

5.2.2. Terms

expression	An expression whose result is type double.
------------	--

5.2.3. Usage

Use the ACOS function to return the arc cosine of an expression.

5.3. ASIN

Returns the arc sine of an expression.

5.3.1. Syntax

ASIN(expression)

5.3.2. Terms

--	--

expression	An expression whose result is type double.
------------	--

5.3.3. Usage

Use the ASIN function to return the arc sine of an expression.

5.4. ATAN

Returns the arc tangent of an expression.

5.4.1. Syntax

ATAN(expression)

5.4.2. Terms

expression	An expression whose result is type double.
------------	--

5.4.3. Usage

Use the ATAN function to return the arc tangent of an expression.

5.5. CBRT

Returns the cube root of an expression.

5.5.1. Syntax

CBRT(expression)

5.5.2. Terms

expression	An expression whose result is type double.
------------	--

5.5.3. Usage

Use the CBRT function to return the cube root of an expression.

5.6. CEIL

Returns the value of an expression rounded up to the nearest integer.

5.6.1. Syntax

CEIL(expression)

5.6.2. Terms

expression	An expression whose result is type double.
------------	--

5.6.3. Usage

Use the CEIL function to return the value of an expression rounded up to the nearest integer. This function never decreases the result value.

x	CEIL(x)
4.6	5
3.5	4
2.4	3
1.0	1
-1.0	-1
-2.4	-2
-3.5	-3
-4.6	-4

5.7. COS

Returns the trigonometric cosine of an expression.

5.7.1. Syntax

COS(expression)

5.7.2. Terms

expression	An expression (angle) whose result is type double.
------------	--

5.7.3. Usage

Use the COS function to return the trigonometric cosine of an expression.

5.8. COSH

Returns the hyperbolic cosine of an expression.

5.8.1. Syntax

COSH(expression)

5.8.2. Terms

expression	An expression whose result is type double.
------------	--

5.8.3. Usage

Use the COSH function to return the hyperbolic cosine of an expression.

5.9. EXP

Returns Euler's number e raised to the power of x.

5.9.1. Syntax

EXP(expression)

5.9.2. Terms

expression	An expression whose result is type double.
------------	--

5.9.3. Usage

Use the EXP function to return the value of Euler's number e raised to the power of x (where x is the result value of the expression).

5.10. FLOOR

Returns the value of an expression rounded down to the nearest integer.

5.10.1. Syntax

FLOOR(expression)

5.10.2. Terms

expression	An expression whose result is type double.
------------	--

5.10.3. Usage

Use the FLOOR function to return the value of an expression rounded down to the nearest integer. This function never increases the result value.

x	CEIL(x)
4.6	4
3.5	3
2.4	2
1.0	1
-1.0	-1
-2.4	-3
-3.5	-4
-4.6	-5

--	--

5.11. LOG

Returns the natural logarithm (base e) of an expression.

5.11.1. Syntax

LOG(expression)

5.11.2. Terms

expression	An expression whose result is type double.
------------	--

5.11.3. Usage

Use the LOG function to return the natural logarithm (base e) of an expression.

5.12. LOG10

Returns the base 10 logarithm of an expression.

5.12.1. Syntax

LOG10(expression)

5.12.2. Terms

expression	An expression whose result is type double.
------------	--

5.12.3. Usage

Use the LOG10 function to return the base 10 logarithm of an expression.

5.13. RANDOM

Returns a pseudo random number.

5.13.1. Syntax

RANDOM()

--

5.13.2. Terms

N/A	No terms.
-----	-----------

5.13.3. Usage

Use the RANDOM function to return a pseudo random number (type double) greater than or equal to 0.0 and less than 1.0.

5.14. ROUND

Returns the value of an expression rounded to an integer.

5.14.1. Syntax

ROUND(expression)

5.14.2. Terms

expression	An expression whose result is type float or double.
------------	---

5.14.3. Usage

Use the ROUND function to return the value of an expression rounded to an integer (if the result type is float) or rounded to a long (if the result type is double).

x	CEIL(x)
4.6	5
3.5	4
2.4	2
1.0	1
-1.0	-1

-2.4	-2
-3.5	-3
-4.6	-5

5.15. SIN

Returns the sine of an expression.

5.15.1. Syntax

SIN(expression)

5.15.2. Terms

expression	An expression whose result is double.
------------	---------------------------------------

5.15.3. Usage

Use the SIN function to return the sine of an expression.

5.16. SINH

Returns the hyperbolic sine of an expression.

5.16.1. Syntax

SINH(expression)

5.16.2. Terms

expression	An expression whose result is double.
------------	---------------------------------------

5.16.3. Usage

Use the SINH function to return the hyperbolic sine of an expression.

5.17. SQRT

Returns the positive square root of an expression.

5.17.1. Syntax

SQRT(expression)

5.17.2. Terms

expression	An expression whose result is double.
------------	---------------------------------------

5.17.3. Usage

Use the SQRT function to return the positive square root of an expression.

5.18. TAN

Returns the trigonometric tangent of an angle.

5.18.1. Syntax

TAN(expression)

5.18.2. Terms

expression	An expression (angle) whose result is double.
------------	---

5.18.3. Usage

Use the TAN function to return the trigonometric tangent of an angle.

5.19. TANH

Returns the hyperbolic tangent of an expression.

5.19.1. Syntax

TANH(expression)

--

5.19.2. Terms

expression	An expression whose result is double.
------------	---------------------------------------

5.19.3. Usage

Use the TANH function to return the hyperbolic tangent of an expression.

6. String Functions

For general information about these functions, see the [Java API Specification](#), [Class String](#). Note the following:

- Pig function names are case sensitive and UPPER CASE.
- Pig string functions have an extra, first parameter: the string to which all the operations are applied.
- Pig may process results differently than as stated in the Java API Specification. If any of the input parameters are null or if an insufficient number of parameters are supplied, NULL is returned.

6.1. INDEXOF

Returns the index of the first occurrence of a character in a string, searching forward from a start index.

6.1.1. Syntax

INDEXOF(string, 'character', startIndex)
--

6.1.2. Terms

string	The string to be searched.
'character'	The character being searched for, in quotes.
startIndex	The index from which to begin the forward search. The string index begins with zero (0).

6.1.3. Usage

Use the INDEXOF function to determine the index of the first occurrence of a character in a string. The forward search for the character begins at the designated start index.

6.2. LAST_INDEX_OF

Returns the index of the last occurrence of a character in a string, searching backward from a start index.

6.2.1. Syntax

LAST_INDEX_OF(expression)

6.2.2. Terms

string	The string to be searched.
'character'	The character being searched for, in quotes.
startIndex	The index from which to begin the backward search. The string index begins with zero (0).

6.2.3. Usage

Use the LAST_INDEX_OF function to determine the index of the last occurrence of a character in a string. The backward search for the character begins at the designated start index.

6.3. LCFIRST

Converts the first character in a string to lower case.

6.3.1. Syntax

LCFIRST(expression)

6.3.2. Terms

expression	An expression whose result type is chararray.
------------	---

6.3.3. Usage

Use the LCFIRST function to convert only the first character in a string to lower case.

6.4. LOWER

Converts all characters in a string to lower case.

6.4.1. Syntax

LOWER(expression)

6.4.2. Terms

expression	An expression whose result type is chararray.
------------	---

6.4.3. Usage

Use the LOWER function to convert all characters in a string to lower case.

6.5. REGEX_EXTRACT

Performs regular expression matching and extracts the matched group defined by an index parameter.

6.5.1. Syntax

REGEX_EXTRACT (string, regex, index)

6.5.2. Terms

string	The string in which to perform the match.
regex	The regular expression.
index	The index of the matched group to return.

6.5.3. Usage

Use the `REGEX_EXTRACT` function to perform regular expression matching and to extract the matched group defined by the index parameter (where the index is a 1-based parameter.) The function uses Java regular expression form.

The function returns a string that corresponds to the matched group in the position specified by the index. If there is no matched expression at that position, `NULL` is returned.

6.5.4. Example

This example will return the string '192.168.1.5'.

```
REGEX_EXTRACT('192.168.1.5:8020', '(.*?)\:(.*)', 1);
```

6.6. REGEX_EXTRACT_ALL

Performs regular expression matching and extracts all matched groups.

6.6.1. Syntax

<code>REGEX_EXTRACT (string, regex)</code>
--

6.6.2. Terms

string	The string in which to perform the match.
regex	The regular expression.

6.6.3. Usage

Use the `REGEX_EXTRACT_ALL` function to perform regular expression matching and to extract all matched groups. The function uses Java regular expression form.

The function returns a tuple where each field represents a matched expression. If there is no match, an empty tuple is returned.

6.6.4. Example

This example will return the tuple (192.168.1.5,8020).

```
REGEX_EXTRACT_ALL('192.168.1.5:8020', '(.*?)\:(.*?)');
```

6.7. REPLACE

Replaces existing characters in a string with new characters.

6.7.1. Syntax

```
REPLACE(string, 'oldChar', 'newChar');
```

6.7.2. Terms

string	The string to be updated.
'oldChar'	The existing characters being replaced, in quotes.
'newChar'	The new characters replacing the existing characters, in quotes.

6.7.3. Usage

Use the REPLACE function to replace existing characters in a string with new characters.

For example, to change "open source software" to "open source wiki" use this statement:
 REPLACE(string,'software','wiki');

6.8. STRSPLIT

Splits a string around matches of a given regular expression.

6.8.1. Syntax

```
STRSPLIT(string, regex, limit)
```

6.8.2. Terms

string	The string to be split.
regex	The regular expression.
Limit	The number of times the pattern (the compiled

	representation of the regular expression) is applied.
--	---

6.8.3. Usage

Use the STRSPLIT function to split a string around matches of a given regular expression. For example, given the string (open:source:software), STRSPLIT (string, ':',2) will return ((open,source:software)) and STRSPLIT (string, ':',3) will return ((open,source,software)).

6.9. SUBSTRING

Returns a substring from a given string.

6.9.1. Syntax

SUBSTRING(string, startIndex, stopIndex)
--

6.9.2. Terms

string	The string from which a substring will be extracted.
startIndex	The index (type integer) of the first character of the substring. The index of a string begins with zero (0).
stopIndex	The index (type integer) of the character <i>following</i> the last character of the substring.

6.9.3. Usage

Use the SUBSTRING function to return a substring from a given string.

Given a field named alpha whose value is ABCDEF, to return substring BCD use this statement: SUBSTRING(alpha,1,4). Note that 1 is the index of B (the first character of the substring) and 4 is the index of E (the character *following* the last character of the substring).

6.10. TRIM

Returns a copy of a string with leading and trailing white space removed.

6.10.1. Syntax

TRIM(expression)

6.10.2. Terms

expression	An expression whose result is chararray.
------------	--

6.10.3. Usage

Use the TRIM function to remove leading and trailing white space from a string.

6.11. UCFIRST

Returns a string with the first character converted to upper case.

6.11.1. Syntax

UCFIRST(expression)

6.11.2. Terms

expression	An expression whose result type is chararray.
------------	---

6.11.3. Usage

Use the UCFIRST function to convert only the first character in a string to upper case.

6.12. UPPER

Returns a string converted to upper case.

6.12.1. Syntax

UPPER(expression)

6.12.2. Terms

expression	An expression whose result type is chararray.
------------	---

--	--

6.12.3. Usage

Use the UPPER function to convert all characters in a string to upper case.

7. Tuple, Bag, Map Functions

7.1. TOTUPLE

Converts one or more expressions to type tuple.

7.1.1. Syntax

TOTUPLE(expression [, expression ...])
--

7.1.2. Terms

expression	An expression of any datatype.
------------	--------------------------------

7.1.3. Usage

Use the TOTUPLE function to convert one or more expressions to a tuple.

See also: [Tuple](#) data type and [Type Construction Operators](#)

7.1.4. Example

In this example, fields f1, f2 and f3 are converted to a tuple.

```
a = LOAD 'student' AS (f1:chararray, f2:int, f3:float);
DUMP a;

(John,18,4.0)
(Mary,19,3.8)
(Bill,20,3.9)
(Joe,18,3.8)

b = FOREACH a GENERATE TOTUPLE(f1,f2,f3);
DUMP b;

((John,18,4.0))
((Mary,19,3.8))
((Bill,20,3.9))
((Joe,18,3.8))
```

7.2. TOBAG

Converts one or more expressions to type bag.

7.2.1. Syntax

```
TOBAG(expression [, expression ...])
```

7.2.2. Terms

expression	An expression with any data type.
------------	-----------------------------------

7.2.3. Usage

Use the TOBAG function to convert one or more expressions to individual tuples which are then placed in a bag.

See also: [Bag](#) data type and [Type Construction Operators](#)

7.2.4. Example

In this example, fields f1 and f3 are converted to tuples that are then placed in a bag.

```
a = LOAD 'student' AS (f1:chararray, f2:int, f3:float);
DUMP a;

(John,18,4.0)
(Mary,19,3.8)
(Bill,20,3.9)
(Joe,18,3.8)

b = FOREACH a GENERATE TOBAG(f1,f3);
DUMP b;

({(John),(4.0)})
({(Mary),(3.8)})
({(Bill),(3.9)})
({(Joe),(3.8)})
```

7.3. TOMAP

Converts key/value expression pairs into a map.

7.3.1. Syntax

TOMAP(key-expression, value-expression [, key-expression, value-expression ...])

7.3.2. Terms

key-expression	An expression of type chararray.
value-expression	An expression of any type supported by a map.

7.3.3. Usage

Use the TOMAP function to convert pairs of expressions into a map. Note the following:

- You must supply an even number of expressions as parameters
- The elements must comply with map type rules:
 - Every odd element (key-expression) must be a chararray since only chararrays can be keys into the map
 - Every even element (value-expression) can be of any type supported by a map.

See also: [Map](#) data type and [Type Construction Operators](#)

7.3.4. Example

In this example, student names (type chararray) and student GPAs (type float) are used to create three maps.

```
A = load 'students' as (name:chararray, age:int, gpa:float);
B = foreach A generate TOMAP(name, gpa);
store B into 'results';

Input (students)
joe smith 20 3.5
amy chen 22 3.2
leo allen 18 2.1

Output (results)
[joe smith#3.5]
[amy chen#3.2]
[leo allen#2.1]
```

7.4. TOP

Returns the top-n tuples from a bag of tuples.

7.4.1. Syntax

TOP(topN,column,relation)

7.4.2. Terms

topN	The number of top tuples to return (type integer).
column	The tuple column whose values are being compared.
relation	The relation (bag of tuples) containing the tuple column.

7.4.3. Usage

TOP function returns a bag containing top N tuples from the input bag where N is controlled by the first parameter to the function. The tuple comparison is performed based on a single column from the tuple. The column position is determined by the second parameter to the function. The function assumes that all tuples in the bag contain an element of the same type in the compared column

7.4.4. Example

In this example the top 10 occurrences are returned.

```
A = LOAD 'data' as (first: chararray, second: chararray);
B = GROUP A BY (first, second);
C = FOREACH B generate FLATTEN(group), COUNT(*) as count;
D = GROUP C BY first; // again group by first
topResults = FOREACH D {
    result = TOP(10, 2, C); // and retain top 10 occurrences of 'second' in
first
    GENERATE FLATTEN(result);
}
```